A First Step Towards Network Security Virtualization: From Concept To Prototype

Seungwon Shin, Haopei Wang, and Guofei Gu

***Abstract*—Network security management is becoming more and more complicated in recent years, considering the need of deploying more and more network security devices/middle-boxes at various locations inside the already complicated networks. A grand challenge in this situation is that current management is inflexible and the security resource utilization is not efficient. The flexible deployment and utilization of proper security devices at reasonable places at needed time with low management cost is extremely difficult. In this paper we present a new concept of *Network Security Virtualization* (NSV), which virtualizes security resources/functions to network administrators/users, and thus maximally utilizing existing security devices/middle-boxes. In addition, it enables security protection to desirable networks with minimal management cost. To verify this concept, we further design and implement a prototype system,** NETSECVISOR**, which can utilize existing pre-installed (fixed-location) security devices and leverage software-defined networking (SDN) technology to virtualize network security functions. At its core,** NETSECVISOR **contains (i) a simple script language to register security services and policies, (ii) a set of routing algorithms to determine optimal routing paths for different security policies based on different needs, and (iii) a set of security response functions/strategies to handle security incidents. We deploy** NETSECVISOR **in both virtual test networks and a commercial switch environment to evaluate its performance and feasibility. The evaluation results show that our prototype only adds a very small overhead while providing desired network security virtualization to network users/administrators.**

***Index Terms*—Security, Software-Defined Networking, Virtualization**

I. INTRODUCTION

WITH the increasing demand of networked services (e.g., e-commerce), network architectures are becoming more and more complicated. For example, a campus network needs to cover diverse departments that have different network interests and security protection requirements/policies, and it causes complex network configuration/management and inefficient usage of network security resources (e.g., pre-installed security devices in certain departments cannot be used for other departments). A cloud network is another good example showing this trend of complex/dynamic network management with diverse security requirements/policies in various internal sub-networks. A typical cloud network commonly consists of a huge amount of hosts (e.g., Amazon has managed nearly half million hosts in 2012 [1]) and network devices to provide services to a large number of dynamic tenants, each having a logically separated network.

This situation is more complicated when there are additional middle-boxes in the network environment. Nowadays, many middle-boxes are employed to improve the performance, robustness, and security of networks. For example, a load balancing proxy server is installed in a network to distribute network flows into target servers, and it is commonly installed into a location where all flows, which should be distributed, are passing. Although these middle-boxes can provide many benefits to networks, they make the network much more complicated to manage [32]. Therefore, several studies have been proposed recently to address this issue [28], [8].

When we consider security for networks (a critical part of network management), the situation is even more complicated. The additional security devices/middle-boxes (e.g., network intrusion detection system and firewall) significantly complicate network configuration/management (e.g., which location to install which device to satisfy the different security needs from different networks while minimizing the overall cost). In addition, security devices have many diverse security functions to serve different purposes. For example, we can use a firewall to control network access, a network intrusion detection system (NIDS) to monitor exploit attacks in network payloads, and a network anomaly detection system to detect DDoS attacks. Therefore, the network administrator should choose reasonable security functions/devices and deploy them into reasonable places. However, it is a tough task for the administrator, because it is hard to predict possible network threats of different network tenants and the administrator is not able to be aware of demands of diverse tenants in advance. Thus, those installed security functions/appliances/devices may not be in the best locations that can best serve the diverse security needs of diverse network users.

Clearly, there is an urgent need to maximize the resource utilization of those existing pre-installed devices/boxes, as well as abstract these security resources to provide a simple interface for network tenants to use (who may not be aware of the exact security device information such as location). Motivated by this problem, we propose a new concept of *Network Security Virtualization* (NSV) that leverages pre-installed, static security devices and provide dynamic, flexible, and on-demand security services to the users. Therefore, users do not need to know the concrete location/number of each kind of security devices/boxes. To realize NSV, we propose two new techniques: (i) transparently controlling flows to desirable

Seungwon Shin is with KAIST, Korea (e-mail: claude@kaist.ac.kr). Haopei Wang and Guofei Gu are with Texas A&M University, College Station, TX, USA (e-mail: haopei@cse.tamu.edu and guofei@cse.tamu.edu).

A preliminary (short) version of this paper appeared in [35]. This material is based upon work supported in part by the Air Force Office of Scientific Research under FA-9550-13-1-0077, the National Science Foundation (NSF) under Grant No. CNS-0954096 and a Google Faculty Research award.

network security services, and (ii) enabling network security *response* functions on a network device.

To maximize the utilization of already installed security middle-boxes, we transparently redirect network flows to desirable security middle-boxes when needed. For example, if a security policy specifies that a network flow should be investigated by a security service, our NSV technology delivers or redirects the flow to the defined security middle-boxes (regardless of its actual physical location) automatically and transparently. Beside this kind of flow controlling, we provide a way of enabling security response function on each network device. Some recent technologies suggest a method to control network flows dynamically at a network device, e.g., Software-Defined Networking (SDN) [24]. With the help of this technology, we can realize some basic security response functions at a network device. For example, we can implement a dynamic access control method at a network device by forwarding or dropping network packets. Extending this technology, we can operate necessary security response functions on a network device when they are required.

To show the feasibility of NSV and verify its ideas, we implement a prototype system for a cloud-like network, in the name of NETSECVISOR.[1] The basic goal of NETSECVISOR is to leverage cloud operators' pre-installed (fixed-location) security devices to provide dynamic, user-friendly, transparent, and on-demand security services to diverse cloud tenants (without professional security knowledge) in a large cloud network with frequent, dynamic VM migrations and configurations. To achieve this goal, we choose to use the most recent Software-Defined Networking (SDN) technology and its most popular realization, OpenFlow [17], [24]. More specifically, we will use SDN/OpenFlow to control the path of traffic (rather than physically adjusting the location of security devices) to leverage pre-installed, *static* security devices to provide *dynamic* service monitoring services. In addition, we enable some basic security response functions, such as network isolation, on network devices.

The design of this prototype, i.e., NETSECVISOR, needs to explore several practical questions. For example, how tenants can express security policies? How are the new paths determined? How are the new paths enforced? How are security response functions enabled? The answers form the main body of this paper.

In short, our contributions can be summarized as follows.

- We propose a new concept of *network security function virtualization* that leverage pre-installed, static security devices to provide dynamic service monitoring services to the network users.
- We implement a prototype system, NETSECVISOR, to enable network security virtualization, as a prototype service to cloud tenants in dynamic cloud networks. In this prototype, we have designed (i) a user-friendly policy script language for both administrators and tenants (Section IV-D), (ii) four routing algorithms (Section IV-E) and (iii) five response strategies (Section IV-F) to address the practical questions raised above.
- We have tested this prototype system in various network environments including virtual networks and a commercial switch environment. The results show that our system adds very few overhead when achieving the desired function.

[1]The principles and techniques of NSV and our NETSECVISOR are applicable to generic networks at any size, as long as there are diverse network security needs from different (sub-)network users/administrators.

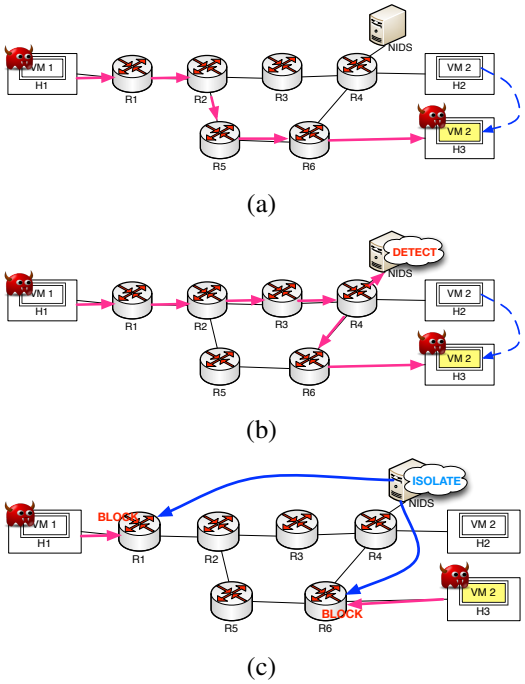


Fig. 1: Example scenario to show how NETSECVISOR works

II. PROBLEM STATEMENT

A. Motivating Example

We start from a simplified motivating example to illustrate the basic idea of NSV. In addition to basic elements such as machines (server/client) and forwarding hardware (switch/router), cloud networks typically consist of a large number of pre-installed (fixed-location) middleboxes and security devices (e.g.,firewall/IDS/IPS) [31], [10]. Our motivating example will create a simple network topology containing these elements and similar to some real cloud networks [4]. As shown in Figure 1, we assume there is a quite simple cloud network consisting of 6 routers (denoted as R1 - R6), 3 hosts (H1 - H3), 2 VMs (VM1 and VM2), and a Network Intrusion Detection System (denoted as NIDS). VM1 lives on H1 and VM2 lives on H2 initially. We also assume that the NIDS is attached to the router R4 to protect VM2 from some network threats. If VM1 in the host H1 is compromised and it tries to infect VM2, this attack will be detected because the NIDS can easily observe the attack from VM1 by mirrored traffic from R4. However, the NIDS can not protect VM2 if it is migrated to host H2 (Fig. 1(a)), which can commonly occur in a cloud network to manage resources. To address this issue, NETSECVISOR dynamically detours network flows to pass through the NIDS (Fig. 1(b)), and thus can still detect

attack toward VM2 (even though it has been migrated to another host) from network threats. In addition, NETSECVISOR enables a security response function to handle infected hosts (here VM1 and VM2). In this scenario, NETSECVISOR isolates infected VMs from a network by blocking network packets from each infected host (shown in (c)).

B. Research Goal and Assumption

The main goal of this work is to propose a new idea, network security virtualization (NSV), and design a prototype system (with the name of NETSECVISOR) that can enable NSV in cloud-like networks to help all tenants easily use security services. More specifically, all tenants do not need to be security experts and do not need to worry about information of security devices (i.e., location, number/kind of devices), operating security functions, and underlying network architecture/workload. Our system mainly guarantees two things; (i) all network packets, which are required to be investigated by a specific security device according to the security policies, are delivered to the right devices in an optimal way and (ii) some basic network security functions are virtualized and operated in network devices supporting SDN like techniques.

Assumption: In NETSECVISOR, we assume our target network has employed SDN/OpenFlow technologies, which means all forwarding devices are SDN/OpenFlow-enabled. We believe that it is a valid assumption, because recent complicated (cloud) networks already began to employ software defined technologies [4], [22], [11], [14]. As SDN is widely considered as the future of networking, we envision that more and more enterprise/campus/home/ISP networks will deploy SDN/OpenFlow technologies in the future. Our solution does not require re-architecting of existing security devices (firewalls, IDS etc.).

III. RELATED WORK

There are some related studies to this work. OpenSafe [3] and Jingling [9] are similar studies to our work. They also provide a script language to monitor networks. Compared with them, our work focuses more on security monitoring in clouds, considers characteristics of different security devices, and designs different routing algorithms and response strategies for security needs. Sekar et al. suggested approaches for NIDS or NIPS deployment [30] through selectively monitoring packets at different nodes and load balancing [12] through traffic replication and aggregation. Raza et al. introduced an approach to route packets to network monitoring points [29]. Even though our work is similar in that our work finds new routing paths for security devices, our work provides in-depth analysis of routing algorithms, and provides more diverse routing and response strategies considering security devices. In a recent study [31], it is shown that it is possible to redirect traffic from Enterprise networks to a cloud service for network processing. However, this study used a relatively complex way to implement that. NETSECVISOR leverages SDN/OpenFlow to simply the routing and provides various flow detouring and response strategies for security purpose.

In [21], [38], researchers discussed how to enforce access control or load balancing policies in an enterprise domain. There are some other studies (e.g. Nettle [37] and Frenetic [20]) that design high level policy languages for OpenFlow networks, which are interesting abstraction solutions. There are already some research achievements on SDN security. FortNOX [26] proposed a new security enforcement kernel and its follow-up work, FRESCO [36], proposed a new modular, composable security application development framework. Avant-Guard [34] implemented two data plane extensions to address both the scalability challenge and the responsiveness challenge which will bring security threat to OpenFlow networks. Rosemary [33] proposed a controller framework design that can protect the control plane from unexpected running failures and losing of network control.

Most recently, a new concept very close to ours is proposed, i.e., Network Function Virtualization [6], which proposes to put network middle-box functions into virtual machines a centralized server farm thus providing efficient network services. While conceptually similar, our proposed NSV differs in several fundamental aspects: First, our work does not need to convert network middle-box functions into virtual machines and relocate into a centralized place, instead it can rely on existing pre-installed security devices and transparently control network flows to desirable devices when necessary. Second, our NSV focuses on providing security virtualization, instead of generic network functions. It has some security specific features, e.g., enabling security response functions and providing a user-friendly security policy script language for both administrators and tenants.

We motivate from the need of flexible management of network security devices/middleboxes. There exist some studies on middle box policy enforcement using SDN [28], [7]. While these studies share some spirit with our work, they are fundamentally different in goals and technical approaches. These studies mainly propose to bind between packets and their "origins" and ensure that packets follow the right sequence of middle boxes. In our work, we focus more on specific security challenges: How to generate the right routing path that leverage pre-installed, static security devices to provide dynamic security services? How to provide a user-friendly way for tenants to specify security policies without worrying about information of security devices (i.e., location, number/kind of devices), operating security functions, and underlying network architecture/workload? Furthermore, how to respond to security alerts? All these are unique in our work. Finally, we note that our approach could potentially benefit from these existing complementary studies [28], [7], particularly if we are going to deal with the service-chaining problem.

IV. DESIGN

In this section, we will describe the architecture and operation scenario of NETSECVISOR.

A. Network Security Virtualization Concept

The main concept of network security virtualization (NSV) is to virtualize network security resources/functions and provide on-demand security to any (possible) networks/places in a

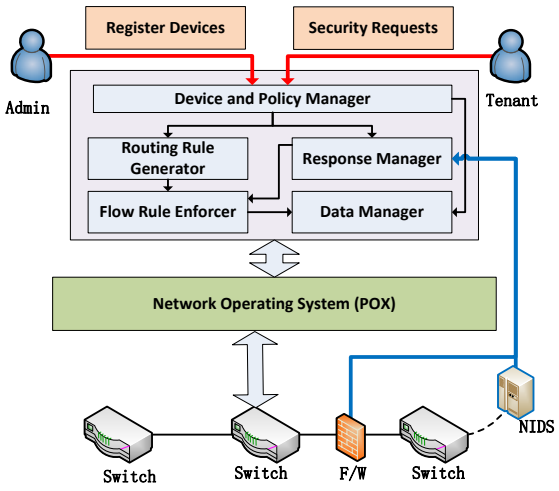


Fig. 2: Overall Architecture

user-friendly way. It requires two main functions: (i) transparently deliver network flows to desirable security devices, and (ii) enable security response functions into a network device when necessary. The first function can allow us to provide security services to any network flow that requires the services, and the second function can help us distribute security services into each network device. Based on these functions, NSV can provide network security services to any network flows at any network devices. In addition, since serving security function is conducted only when necessary, we can efficiently manage network security resource usage.

B. Overall Architecture of NETSECVISOR

It is not easy to realize network security virtualization (NSV) with traditional network technology because it lacks several features, such as network-wide monitoring, network configuration, network flow control, and response management. To address this issue, we leverage an emerging network technique, Software-Defined Networking (SDN) [24], which can help us dynamically control network flows and monitor whole network status easily. And we have implemented a prototype system, NETSECVISOR (NSV services for a cloud-like network) based on SDN. As shown in Figure 2, NETSECVISOR consists of five main modules: (i) Device and policy manager, (ii) Routing rule generator, (iii) Flow rule enforcer, (iv) Response manager, and (v) Data manager.

Device and policy manager is in charge of two main functions. First, it receives the information of security devices from a cloud administrator, and it stores that information into a *device table* in NETSECVISOR for further usage. Second, this module also receives security requests from each network tenant, and it translates them into security policies and stores the policies into a *policy table*. Thus, this module finally has two information: (i) locations/types of security devices from a cloud administrator and (ii) security policies from each tenant. It makes our system handle network security devices easily.

Response manager receives detection results from security devices, and it enables security response strategies that are defined in security policies, when it is needed. For example, if a tenant defines a security policy to drop all corresponding packets when a threat is detected by a NIDS, the response manager will enable *drop* function to discard network packets belonging to the detected network flows on a network device. Enabled functions will be realized as a set of network flow rules, which are sent to routers or switches, and thus we can leverage each network device as a kind of security device (e.g., firewall).

Routing rule generator creates routing paths to control each network flow. When creating routing paths, this module investigates security polices (from each tenant) to satisfy their requirements. For example, if a tenant defines a security policy that specifies all network flows to port 80 should be inspected by a NIDS attached to a router A, then this module produces (a) routing path(s) to let all network packets heading to port 80 pass through the router A. It helps our system assign security requirements to each security device based on efficiency (in terms of security resource management) and effectiveness (in terms of finding reasonable security devices).

Flow rule enforcer enforces flow rules to each OpenFlow router and switch. If the response manager enables response strategies or the routing rule generator produces routing paths, this module translates them into flow rules that could be understood by OpenFlow routers/switches. After translation, it sends translated rules to corresponding routers or switches.

Data manager captures network packets from routers or switches to hold until some security devices send their detection results to NETSECVISOR. The reason why it holds packets is to enable some in-line style security funtions as what generic Intrusion Prevention Systems provide. This module does not hold packets all the time, but only captures and stores when necessary (i.e., a security policy specifies an in-line mode action for response).

A typical operation of NETSECVISOR works as follows. A network administrator registers network security devices (both physical devices and virtual appliances) to NETSECVISOR. After registration, cloud tenants need to create their security requests and submit them into NETSECVISOR. Then, NETSECVISOR parses the submitted security requests to understand the intention of tenants and writes the corresponding security policies to policy table. Next, if NETSECVISOR receives a new flow setup request from a network device, it checks whether this flow is matched with any submitted policies. If it is, NETSECVISOR will create a new routing path and corresponding flow rules for the path. At this time, NETSECVISOR guarantees that the routing path includes required security devices that are defined in a matched policy (i.e., the first NSV function). After this operation, it enforces flow rules to each corresponding network device to forward a network flow. If any of security devices detects malicious connection/content from monitored traffic, they will report this information to NETSECVISOR. Based on the report and submitted policies, NETSECVISOR enables a security response function to respond to malicious flows accordingly (i.e., the second NSV function).

C. How to Register Security Devices

To use existing security devices, a cloud administrator needs to register them to NETSECVISOR using a simple script language. The script language asks for the following information in registration: (i) device ID, (ii) device type (e.g. firewall and IDS), (iii) device location (e.g., attached to a router A), (iv) device mode (passive or in-line), (v) supported functions (e.g., detect HTTP attacks).

Here is an example scenario to use the script. Let's assume that we install an intrusion detection system (IDS) into a cloud network, and the IDS is attached to a network switch whose data path ID is 089. In addition, we assume that this IDS is operated in passive mode, and it protects a network by detecting DNS attacks. This IDS can be registered to NETSECVISOR by using the following script: {1, IDS, 089, passive, detect DNS attack}

D. How to Create Security Policies

After a network administrator registers security devices for a cloud network to NETSECVISOR, the information of the registered security devices is shown to tenants using the cloud network by NETSECVISOR. Then, the tenants can define their security requests considering registered security devices and security functions enabled by NETSECVISOR (these enabled functions will be presented in section IV-F). Motivated from [5], security request of a tenant is described with a script language that NETSECVISOR provides. The script for a request consists of 3 fields: (i) flow condition, which specifies the flow to be monitored (or controlled), (ii) function set, which denotes the necessary security devices for monitoring or investigating, and (iii) response strategy, which defines how to handle the flow if a threat is detected. The policy syntax is: {{*flow condition*}, {*function-list*}, {*action-list*}}. Currently, NETSECVISOR supports 5 different response strategies (two in passive mode and three in in-line mode), and they are *drop, isolate* for passive mode and *drop, isolate, redirect* for in-line mode. Detailed information for these responses is explained in Section IV-F. Here, we provide an example script for the following security request: one tenant (IP = 10.0.0.1) wants all HTTP traffic regarding to his IP to be monitored by a firewall and an IDS, and it wants to drop all packets issued as attacks by the firewall and the IDS. This request can be sent to NETSECVISOR with the following script: {{((DstIP = 10.0.0.1 OR SrcIP = 10.0.0.1) AND (DstPort = 80 OR SrcPort = 80))}, {firewall, IDS}, {drop}}

Finally, NETSECVISOR receives security requests from each tenant, and it translates them into security policies that can be applicable to a SDN enabled cloud network. At this time, NETSECVISOR needs to translate tenant defined high-level conditions into more specific network level conditions, and it also maps function set into security devices registered before.

E. How to Decide Routing Path

If NETSECVISOR finds network packets meeting a flow condition specified by a security policy, then it will route these packets to satisfy security requirements. When NETSECVISOR routes network packets, it should consider the following two conditions: *(i) network packets should pass through specific security devices to meet the security requirements, and (ii) the created routing paths for network packets should be optimized*.

There are several existing routing algorithms for intra-domain (e.g., OSPF [13]) to find optimal paths. However, they can not be employed directly for our case. Since network packets only contain the source and destination information, existing routing approaches can not discover necessary ways to locations where security devices are installed. Thus, we need to create our own approaches.

What kinds of basic properties should be in our routing algorithms? Described in Section IV-D, NETSECVISOR supports two modes of security devices which are passive mode and in-line mode. For a passive mode device, we can forward the traffic to pass through the device, or just mirror a duplicate to the device and forward the original traffic in another way. For an in-line mode device, all traffic should pass through and be monitored by this device. The generated routing path should satisfy the requirements from different modes of security devices. Besides, a network may contain only passive mode devices or in-line mode devices, or both the two kinds. Thus, we aim to design different algorithms for different usage scenarios.

Recent software-defined networking technologies (e.g., OpenFlow) provide several interesting functions, and one of them is to control network flows as we want. With the help of this function, we propose 4 different routing algorithms, which can satisfy different requirements. We define the following 4 terms to explain our algorithms more clearly: (i) *start node*, a node sends network packets, (ii) *end node*, a node receives the packets, (iii) *security node*, a node mirror packets to a passive security devices, and (iv) *security link*, a link on which in-line security devices are located. Among the proposed 4 algorithms, 3 of them (i.e., Algorithm 1 - 3) are designed for security policies which only use passive mode devices, and 1 of them (i.e., Algorithm 4) is suggested for policies which have in-line security devices such as a firewall and a NIPS.

To describe our algorithms more clearly, we first explain how we can find the path between two nodes.

A network can be characterized using a *graph structure*, which consists of nodes (hosts or routers or switches) and arcs (physical links between devices). In this graph structure, we need to find some paths between a start node, which sends network packets, and an end node, which receives network packets. At this time, we usually want to find the shortest path[2] between a start node and an end node to deliver network packets efficiently. The problem of finding the shortest path between two nodes is a type of a linear programming, and it can be formulated as the minimum cost flow problem [16]. To do this, we first need to define some variables: $x_{i,j}$, which represents the amount sent along the link from node i to node j, and b_i, which means the available supply at a node (if $b_i \leq 0$, then there is a required demand). In addition, we assume

[2]Here, the shortest path means that the path represents the lowest network link cost, and the network link cost can be determined by several features, such as network capacity and current load.

that a network is balanced in the sense that $\sum_{i=1}^{n} b_i = 0$. Considering the unit cost for flow along the arc between two nodes i and j as $c_{i,j}$, the minimal cost flow problem can be formalized as the following mathematical terms in Eq. (1).

$$\begin{aligned} &\min \sum c_{i,j} x_{i,j} \\ \text{s.t} \sum_{j=1}^{n} x_{i,j} - \sum_{k=1}^{n} x_{k,i} &= b_i \text{ for i = 1, 2, ... n} \\ x_{i,j} &\geq 0 \text{ for i,j = 1, 2, ... , n} \end{aligned} \tag{1}$$

Based on this Eq. (1), we can find the shortest path between two nodes[3]. We will use this result as a primitive to find paths satisfying the conditions in our problem domain. For brevity, when we find the shortest path between a and b, we denote Eq. 1 as *find_shortest_path(a, b)*.

To describe the proposed algorithms more clearly, we will provide concrete examples to illustrate the key concept of each algorithm. For the illustration, we use a simple network structure as shown in Figure 3(a). It contains six routers (R1 - R6), a start node (S), an end node (E), and a security device (C) attached to node R4 (thus R4 is a security node). We assume that node S sends packets to node E, and our example security policy is specified that all packets from node S to node E should be inspected by security device C. Furthermore, Figure 3(b) shows the traditional packet delivery based on the shortest path routing *without* considering the need of security monitoring. Thus, packets from node S are simply delivered through the path of (S $\rightarrow$ R1 $\rightarrow$ R5 $\rightarrow$ R6 $\rightarrow$ E), and obviously in this case they can *not* be inspected by the security device C. Next we will describe how our new algorithms work and illustrate them on the same network structure.

Algorithm 1 (Multipath-Naive): First, we design a simple algorithm to visit each security node regardless of the path between a start node and an end node. In this algorithm, NETSECVISOR first finds the shortest path between a start node and an end node. Then, NETSECVISOR also discovers the shortest paths between a start node and each security node. If NETSECVISOR has all paths, it delivers packets to all obtained paths. This approach is based on a function of OpenFlow, which can send network packets to multiple output ports of a router. Thus, NETSECVISOR can send network packets to different paths simultaneously. This approach is summarized in Algorithm 1. An example case for this algorithm is shown in Figure 3 (c), and here, this algorithm finds the shortest path between S and E for a packet delivery and the other shortest path between S and R4 (S $\rightarrow$ R1 $\rightarrow$ R2 $\rightarrow$ R3 $\rightarrow$ R4) for inspection.

Algorithm 2 (Shortest-Through): The second approach is to find the shortest path between a start node and an end node passing through each intermediate security node. Finding this path is more complicated than finding the shortest path between two nodes, because in this case, we should make sure that the found path include all intermediate nodes. To do this, NETSECVISOR finds all possible connection pairs (e.g., if there are multiple security nodes, (a start node, a security node 1) and (a security node 1, a security node 2)) among all nodes including a start, an end, and security nodes, and then, it investigates the shortest paths of each pair. After this operation, it checks possible paths between a start node and an end node, and it could generate multiple paths. Finally, NETSECVISOR finds the path which has the minimum cost value. This approach is formalized in Algorithm 2, and an example case is presented in Figure 3 (d). In this case, it finds the shortest path between S and E that passes through R4, and the path is like the following (S $\rightarrow$ R1 $\rightarrow$ R2 $\rightarrow$ R3 $\rightarrow$ R4 $\rightarrow$ R6 $\rightarrow$ E).

[3]We do not talk about how we can solve this problem in this work, because it is well-known [16], and it is not our main focus.

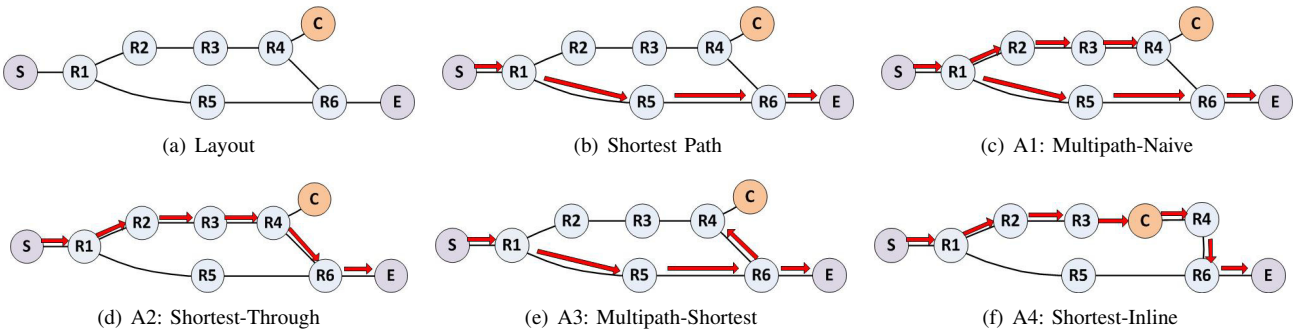
Algorithm 1: multipath-naive

Input: S (start node)
Input: E (end node)
Input: C_i = security node i, $i = 1, 2, 3, .., n$
Output: FP_m (multiple shortest paths)
$P_0 \leftarrow$ find_shortest_path(S, E);
foreach C_i **do**
  $P_i \leftarrow$ find_shortest_path(S, C_i);
foreach P_i **do**
  if $P_i \not\subset P_0$ **then**
    $FP_m \leftarrow P_i$;
  foreach P_j **do**
    if $i \neq j$ *and* $P_i \not\subset P_j$ **then**
      $FP_m \leftarrow P_i$;

Algorithm 2: shortest-through

Input: S (start node)
Input: E (end node)
Input: C_i = security node i, $i = 1, 2, 3, .., n$
Output: FP, one shortest path
foreach C_i **do**
  $M_i \leftarrow C_i$;
$M_i \leftarrow S$;
$M_i \leftarrow E$;
foreach M_j **do**
  foreach M_l **do**
    if $M_j \neq M_l$ **then**
      $P_{j,l} \leftarrow$ find_shortest_path(M_i, M_l);
while $R_k \neq NULL$ **do**
  $R_k \leftarrow$ permutation(M_1, M_1, ..., M_{n+1}, M_{n+2},);
  if *first element of* R_k = S **then**
    if *last element of* R_k = E **then**
      $Q_t \leftarrow R_k$;
foreach Q_t **do**
  foreach $P_{j,l}$ **do**
    if $P_{j,l} \subset Q_t$ **then**
      $TP_t \leftarrow P_{j,l}$
FP = min($TP_1, TP_2, ...$);

Algorithm 3 (Multipath-Shortest): As we mentioned previously, OpenFlow supports the function of sending out network packets to multiple outports of a router simultaneously, and Algorithm 1 is based on this function. However, it may not be efficient, because it can create multiple redundant network flows. Thus, we try to propose an enhanced version



(a) Layout (b) Shortest Path (c) A1: Multipath-Naive

(d) A2: Shortest-Through (e) A3: Multipath-Shortest (f) A4: Shortest-Inline

Fig. 3: Example Scenario for Each Algorithm

of Algorithm 1. The concept of this enhanced algorithm is similar to that of Algorithm 1. However, this approach does not find the shortest path between a start node and each security node, instead it finds a node, which is closest to a security node and in the shortest path between a start node and an end node. If it finds the node, it asks this node to send packets to multiple output ports: (i) a port, which is connected to a next node in the shortest path, and (ii) (a) port(s), which is (are) connected to (a) node(s) heading to (a) security node(s). Thus, network packets are delivered through the shortest path, and they are delivered to each security node as well. This approach is presented in Algorithm 3. Figure 3 (e) presents an example scenario for this algorithm. It first finds the shortest path between S and E, and it finds the shortest path between R4 and nodes on the found shortest path, which is R6 $\rightarrow$ R4.

Algorithm 3: multipath-shortest

```
Input: S (start node)
Input: E (end node)
Input: C_i = security node i, i = 1, 2, 3, .., n
Output: FP_j, multiple shortest paths)
P_0 = find_shortest_path(S, E);
FP ← P_0;
foreach C_i do
    foreach n_j in P_0 do
        TP_{i,j} ← find_shortest_path(C_i, n_j);
        FP ← TP_{i,j};
```

Algorithm 4: shortest-inliine

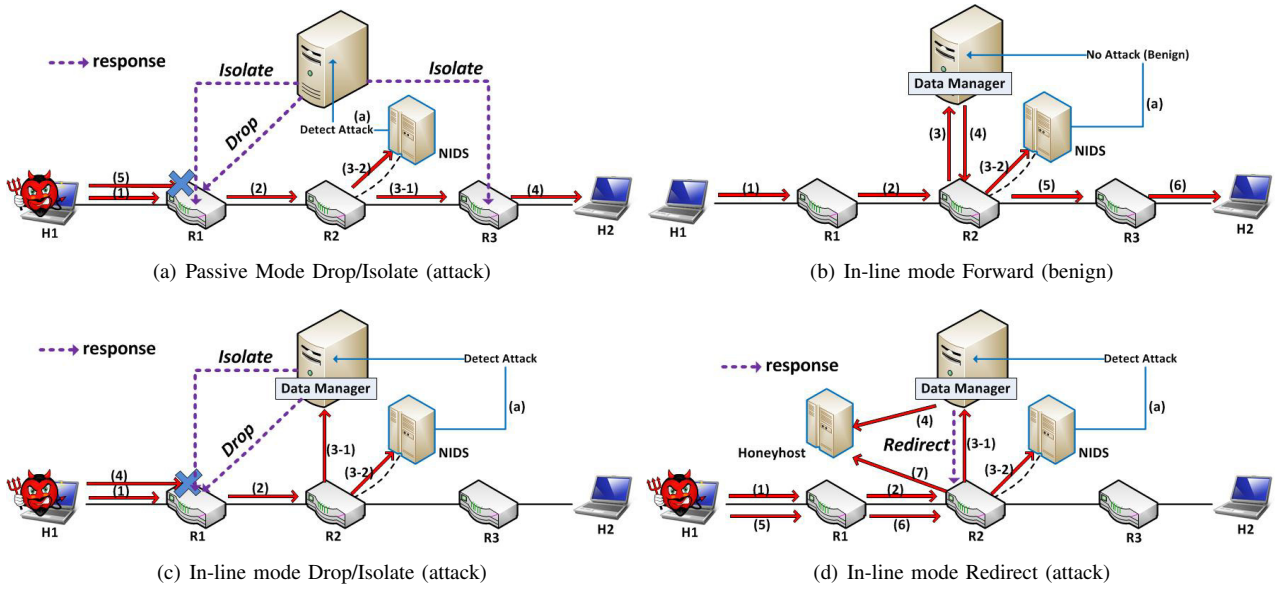
```
Input: S (start node)
Input: E (end node)
Input: C^i_{m,n} = security link i between m node and n node
Output: FP, one shortest path
foreach C_i do
    M_i ← C_i;
M_i ← S;
M_i ← E;
foreach M_j do
    foreach M_l do
        if M_j ≠ M_l then
            P_{j,l} ← find_shortest_path(M_j, M_l);
while R_k ≠ NULL do
    R_k ← permutation(M_1, M_1, ..., M_{n+1}, M_{n+2},);
    if first element of R_k = S then
        if last element of R_k = E then
            foreach C^i_{m,n} do
                if (m, n) ⊄ R_k then
                    continue;
                Q_t ← R_k;
foreach Q_t do
    foreach P_{j,l} do
        if P_{j,l} ⊂ Q_t then
            TP_t ← P_{j,l}
FP = min(TP_1, TP_2, ....);
```

Algorithm 4 (Shortest-Inliine): Previously, we have presented three algorithms, and they are applicable to the case, if security devices attached to security nodes are passively monitor network packets. However, if a security device is installed as in-line mode, the situation should be changed. For passive monitoring devices, we can simply find a path passing through each security node, however, in the case that there is a security device working in-line mode, we are required to consider both of security nodes and security links (between two nodes). Even though a path includes two nodes for a link, it does not guarantee that the link is used for the path, because each node could be linked to another nodes. To address this issue, we modify our Algorithm 2 to make sure that it should include security links in the generated path. Thus, this Algorithm 4 has a routine checking whether security links are included or not. This algorithm is summarized in Algorithm 4. An example for this case is presented in Figure 3 (f), and it shows that there is a security node (C) on the link between R3 and R4. The selected path is the same as the path found in Algorithm 2. However, to find a path considering an inline device, we need to make sure that the link, where an inline device is, is on the routing path.

Usage Scenario and Comparison of Each Algorithm: Finally, we have compared each algorithm and presented their pros/cons and suitable using scenarios in Table I. Understanding strong or weak points of each algorithm will help us find a more suitable routing algorithm for specific situation in a cloud network environment. Table I summarizes the strong or weak points of each algorithm, as well as the recommended scenario to use each algorithm. For example, we can see that the advantage of Algorithm 2 (Shortest-Through) is that it will

Algorithm	Pros	Cons	When to use
A1: Multipath-Naive	Simple and fast	Redundant flows	Enough network capacity, delay is important
A2: Shortest-Through	No redundant path	Computation overhead, when multiple devices	Not enough network capacity, delay is not so important
A3: Multipath-Shortest	Efficient routing path	Computation overhead	Not many hops (e.g., communication between inside VMs)
A4: Shortest-Inline	Guarantee passing through a specific link	Computation overhead, when multiple devices	For an inline security device(e.g., IPS)

TABLE I: Comparison of Each Algorithm



(a) Passive Mode Drop/Isolate (attack)

(b) In-line mode Forward (benign)

(c) In-line mode Drop/Isolate (attack)

(d) In-line mode Redirect (attack)

Fig. 4: Response Strategies

not increase much network traffic and the disadvantage is its relatively high running complexity (actually it has the highest computation complexity among four). It is mostly suitable to use if the overall network capacity is a concern while the communication delay is not a concern. Algorithm 1 (Multipath-Naive) is suitable for cloud networks with enough capacity, and their communication delay is of importance. Algorithm 3 (Multipath-Shortest) is mostly suitable for relatively short paths without many hops. Algorithm 4 (Shortest-Inline) is obviously suitable for inline security devices. In fact, if the users want to use in-line devices, our system will automatically choose Algorithm 4 for them. In general, the system allows the users to specify/configure the algorithms they want to use. Also, the system can automatically choose an algorithm for users if the users simply specify their high-level priorities (e.g., communication or computation cost), based the summary in Table I.

F. How to Enable a Security Response Function

After inspecting network flows with specified security devices, a tenant can decide response methods for network packets detected as malicious (or suspicious) by security devices. He may want to drop all detected packets or isolate some infected hosts. These response methods are very hard to be implemented with existing network devices, because it requires additional inline devices that can handle network packets or changing the configurations of a network device. For example, suppose that we install several intrusion detection systems into a cloud network, in this case, what if a tenant wants to drop all network packets detected by these intrusion detection systems? Since these systems cannot drop packets, we need to have another methods (e.g., additional proxy devices) to support the tenant's request.

To address this issue and provide more flexible response methods, NETSECVISOR provides a way of enabling **5 security response strategies**, and they do not require adding physical security devices or changing network configurations for handling packets. These methods can be operated into two different modes: (i) passive mode, and (ii) in-line mode. *Passive mode response strategies* are similar to strategies by existing network intrusion detection systems that mirror network traffic for investigation and generate alerts. In this mode, some malicious network traffic could have been already delivered to a target host.

In this passive mode, NETSECVISOR supports two response strategies. First, NETSECVISOR can *drop* packets that belong to detected network flows. This strategy is useful to stop some later malicious packets in the flow, but it does not guarantee that none of malicious packets are delivered to the target host. Second, NETSECVISOR can *isolate* a specific host or a VM, if it is detected as malicious. In this strategy, NETSECVISOR is able to block sending network packets to a detected host or

a VM, or from a detected host or a VM. A tenant can specify which kind of packets should be blocked (i.e., packets from, packets to, and both). The operation of these two strategies are shown in Figure 4 (a). If a host H1 sends a malicious packet (1) to a host H2, this packet will be mirrored by a router (3-2) and delivered to a NIDS. Then, NIDS detects this as malicious and reports to NETSECVISOR (a). However, since NIDS is installed in passive mode (mirror network traffic), it does not interfere network flows, thus, the packet is also delivered to a target host H2 (3-1) (4). If NETSECVISOR receives an alert from NIDS, NETSECVISOR will enforce flow rules to router R1 for dropping packets from H1 (drop strategy) or enforce flow rules to router R1 (or R3) for dropping any packets from/to H1 (or H2)[4] (isolate strategy).

NETSECVISOR also provides three interesting in-line mode response strategies: (i) *drop*, (ii) *isolate*, and (iii) *redirect*. In the case of in-line strategies, NETSECVISOR holds network packets until security devices send their decision results. If NETSECVISOR receives detected results and the response strategy for these packets is in-line mode drop, it discards all packets. Thus, a host, which an attacker tries to infect, does not receive any malicious packets. This action is denoted in Figure 4 (c). In this case, a malicious host H1 sends an attack packet (1) to H2, then the packet is delivered to a NIDS (3-2) and hold by data manager module (3-1). If NIDS decides the packet is malicious (a), NETSECVISOR discards held packets, and it also enforces flow rules to a router for drop/isolate strategy. If NIDS considers the packet is benign (this scenario is depicted in Figure 4 (b)), the held packets will be forwarded to a target host (H2) automatically (4) and (5). Both drop and isolate strategies are similar to passive mode except the in-line mode guarantees that no malicious packets are delivered to victims. Redirect strategy is different from drop/isolate, and it does not drop packets but detours packets to another networks or hosts. To handle malicious packets, we sometimes employ some honeyhost, which is a decoy host imitating a normal one. Redirect strategy can be used to redirect malicious traffic to the honeyhost transparently (without changing any network configurations or applications). Like other in-line mode strategies, redirect strategy also holds packets in data manager, and if security devices detect some malicious packets, they will be redirected to another host. Figure 4 (d) shows this scenario. Here, we can see that the held packets (4) and detected packets (7) are redirected to a honeyhost for further investigation.

V. IMPLEMENTATION

Our prototype is implemented on top of the POX controller [27], a popular lightweight controller system for Openflow networks. NETSECVISOR contains approximately 1,200 lines of python code. The two data structures (device table and policy table) in the device and policy manager are implemented as simple hash tables. The response manager is implemented as a simple network server and it opens a network connection to receive detected results from each security device. The data table in the data manager is a simple hash table, whose key is created from network 5-tuple information (i.e., source/destination IP address, source/destination port, and protocol) of a network flow. The routing rule generator firstly uses the topology discovery component in POX to learn the underlying topology of the network as a graph structure. It also collects network status information to estimate cost of each network link through periodically sending query through POX APIs. Finally, the flow rule enforcer translates the routing rules into flow rules and sends the flow rules to relevant switches through POX APIs (e.g., `ofp_flow_mod`).

[4]We can regard that H2 is also infected, then, we may want to isolate H2 as well.

VI. EVALUATION

A. Evaluation Environment

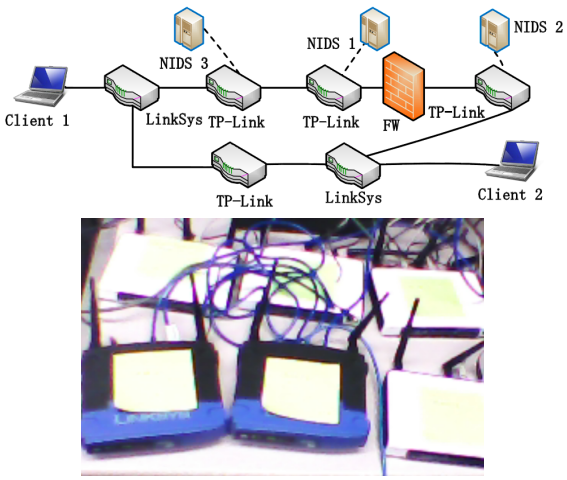
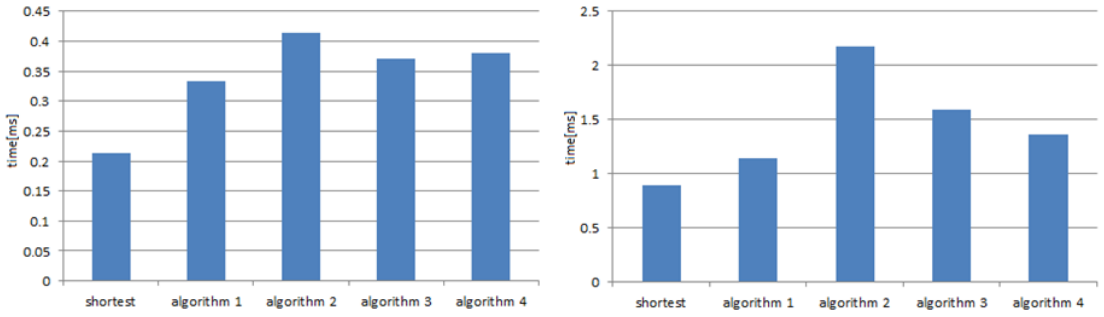


Fig. 5: 6-router Testbed

To verify the feasibility and evaluate the efficiency of NETSECVISOR, we emulate three different network topologies, and two topologies are running on a virtual network environment and one topology is running on a real commercial switch environment.

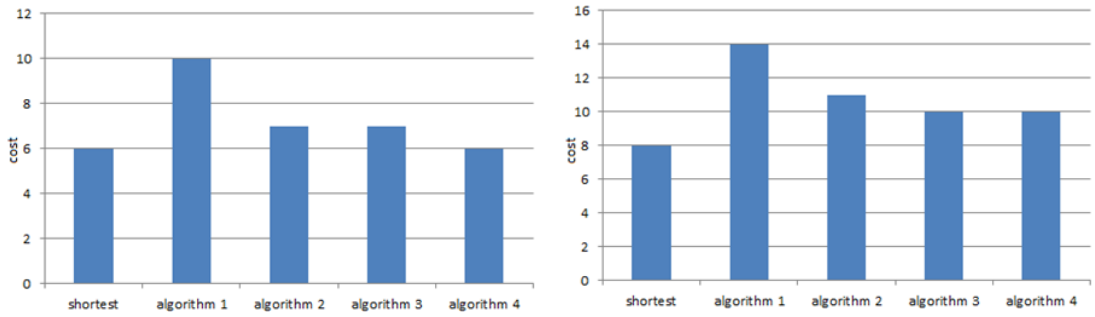
Virtual network environment: We select Mininet [18], which is popularly used for emulating OpenFlow network environments, to emulate two different network topologies: (i) 12-router configuration and (ii) 64-router configuration. In each case, we install one passive mode security device and one in-line mode security device (for evaluating Algorithm 4). In addition, we create two hosts for a start node and an end node for packet traversal.

Commercial switch environment: We create a 6-router network topology presented in Figure 5 with 6 OpenFlow-enabled switches (2 LinkSys WRT54GL switches and 4 TP-Link TL-WR1043ND switches), 3 hosts for NIDS, and 2 hosts for a client and a server. We change the firmware of LinkSys and TP-Link devices to enable OpenFlow functions [25], and we install Snort for NIDS. In the current status, we only create a 6-router network because of the hardware limitation. However, we will test more diverse network topologies with more real network devices in near future.



(a) 12 routers (b) 64 routers

Fig. 6: Flow Rule Generation Time Measurement



(a) 12 routers (b) 64 routers

Fig. 7: Network Cost Measurement

B. Generation Time and Network Cost Measurement

We measure four metrics to estimate the performance overhead of NETSECVISOR. First, we measure the routing path generation time of each algorithm. It is an important feature to estimate the performance of NETSECVISOR, because it determines how many network flows can be handled by NETSECVISOR. For example, if NETSECVISOR can create a routing path in 1 ms, it means that NETSECVISOR can handle 1,000 network flows in 1 second. Of course, there are many ways to increase the number of flows for handling, thus, it does not mean that the routing path generation time strictly restricts this number. Second, we estimate the network cost that represents total cost of delivering a packet between a start node and an end node, and this cost can be formalized as the following formula: $\sum_{i,j \in M} c_{i,j}$, where $c_{i,j}$ is the unit cost for flow along the arc between two nodes i and j, and i, j are pairs of nodes belonging to a path M. When we measure this network cost, we set $c_{i,j}$ as 1, but in real cases, we can define this cost by other methods (e.g., measuring network link status). In our evaluation, we measure the number of network links between a start node and an end node after generating the flow rule as the network cost. Third, we measure the CPU and memory overhead of NETSECVISOR, when it determines a routing path. Fourth, we measure the average response time between a client host and a server host, when NETSECVISOR sets up a routing path between them. In this case, we want to understand the response time when a client sends packets to a server with different routing paths. When we measure each metric, we compare our routing algorithms with an simple baseline routing module (except CPU and Memory overhead case). This module is based on the Dijkstra's shortest path algorithm, which is well known and widely used, and we denote this algorithm as *shortest* in our test results. Based on the comparison, we can estimate the overhead of the proposed routing algorithms.

Virtual network environment: The results for the routing path generation time are shown in Figure 6, and we can observe that the proposed routing algorithms add relatively high overhead (from 20% to 100%) compared with the baseline module. However, we argue that the time is still reasonable because it can still handle around 500 network flows per second in the worst case (i.e., running Algorithm 2 in 64-router configuration), and it can be improved by optimized implementation (all test results are based on our unoptimized prototype system). An interesting thing is that the added relative overheads (in terms of percentage) are decreasing when we apply more routers. For example, in the case of 12-router configuration, Algorithm 1 shows nearly 50% of overhead, but it is reduced into around 20% when we use 64 routers. This is probably because in the case of larger networks, finding the shortest path between nodes (the primitive for both baseline and our algorithms) consumes most of time for routing path calculation (instead of our new routing calculation). It implies that if we apply NETSECVISOR into a large-scale cloud network, the overhead could be reduced significantly compared with the based module. Figure 7 presents the test results of

network cost for each case. Here, we find that Algorithm 1 adds much more overhead than other cases, and it is natural because this algorithm copies network flows to be monitored. However, other algorithms add small overhead, because they try to find the shortest paths for their purpose.

Commercial switch environment: Figure 8 presents the test results of the routing path generation time (a) and the network cost (b) for the commercial switch environment. In this test, we have changed the number of security devices from 1 to 3 to understand the overhead more clearly (In the figure, 1 location denotes that we install 1 security device). In the case of the routing path generation time, the overhead is somewhat noticeable, which is an expected result from the virtual network environment case. An interesting thing from the results is that even though we install more security devices into a network, it does not cause much serious overhead. Of course, it is possible that the number of routers for testing is limited (i.e., 6), and thus installing more security devices may not produce complicated path calculations. We present the test results of network cost in Figure 5 (b), and it clearly shows that Algorithm 1 adds more overhead than other algorithms, which is the same result that we found in the above.

C. CPU and Memory Overhead and Response Time

CPU and memory overhead for virtual switch environments: We also evaluate the CPU and memory usage overhead in each topology. The overhead is introduced when NETSECVISOR generates routing paths. The baseline is in idle state and the usage of CPU and memory is zero. For different topologies, we calculate the absolute increment of usage when generating routing paths compared to this baseline. Since we find that the CPU and memory usage overhead of each algorithm case is quite close to each other, we use the result of Algorithm 4 as a representative example. The overhead results are shown in Figure 9, and we can clearly see that the memory overhead is no more than 1%, which can be ignorable. In addition, our routing algorithms do not produce serious CPU overhead (the 64-router topology only adds about 6% overhead). The results imply that we can optimize our algorithms by using more CPU power and memory. For example, we can employ some techniques to hold all (or most of) created routing paths into a memory space to reduce the time for routing path generation (a kind of cache).

Average response time for commercial switch environment: To measure the average response time between two peers (i.e., Client 1 and Client 2 in Figure 5 (left)) for each algorithm, we first set up routing paths between two peers with each algorithm. After generating the routing paths, we send 60 `ping` from Client 1 to Client 2 to measure the average response time. In this case, the routing paths for each algorithm are the same as the paths in Figure 3, and the test results are shown in Figure 10. The results clearly show that if we use Algorithm 1 or Algorithm 3, a client cannot feel any additional delay, and that is because these two algorithms guarantee that they deliver packets from the client to the server through the shortest path. Other two algorithms (2 and 4) add a little more delays compared with other methods, and it is very natural because they take longer paths.

D. Discussion on Scalability

We aim to design NETSECVISOR to scale to cloud level network. A large cloud network often contains millions of tenants and VMs. That introduces a data plan scalability issue. There may be tens of thousands of security requests generated per second. Generating routing paths for all of them can be a huge burden for NETSECVISOR. A single instance of NETSECVISOR may not be able to handle that. Meanwhile, the inherent bottleneck of the flow rules amount in the data plane is another possible limitation. We need some optimization methods to make our system scale well. To solve this issue, we could leverage some existing research solutions. To address the bottleneck of generating routing paths, we notice that generating of each routing path can be done separately and asynchronously, which means we could run our system in a distributed manner with the help of existing distributed SDN controller platforms (e.g., Onix [15], ONOS [23]) to generate the routing path. We could have multiple NETSECVISOR working in parallel. Onix and ONOS both provide suitable distribution solutions. Thus, we think separately and asynchronously generating of routing paths for different policies could be a potential solution to the scalability challenge. To address the bottleneck of data plane flow entries, we could implement scalable and efficient networking system (e.g., DIFANE [19]) in some heavily loaded nodes (e.g. aggregation switches, ToR switches).

E. Case Study

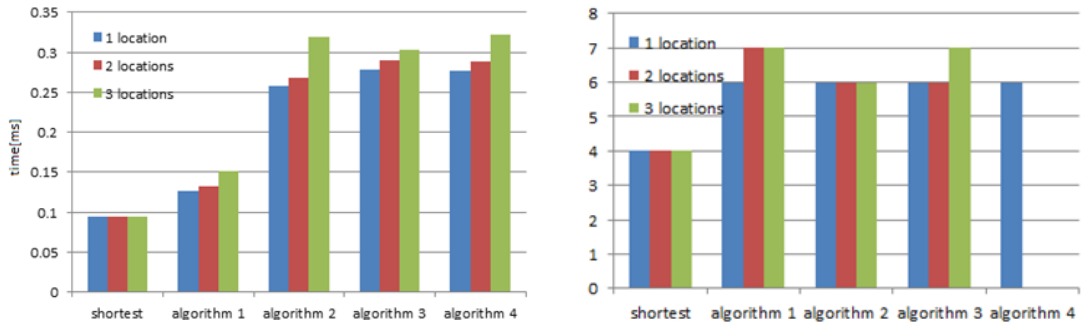
In addition, we show a concrete case study of using NETSECVISOR to enforce a security policy to respond to certain network threats. We set up a test case to present how a tenant can respond to malicious network packets with NETSECVISOR, and this test has been conducted in the commercial switch environment (Figure 5 in Appendix). We generate a simple DoS attack (exploiting the Real Audio server view-source DoS vulnerability) from Client 1 to Client 2.

A sample security requirement from a tenant in this network is to drop all packets detected by the NIDS device, and thus, it registers a simple security policy $\{\{ALL\}, \{NIDS\}, \{drop\}\}$. NETSECVISOR uses Algorithm 1 to generate the routing paths and uses Snort as the security device. When launching this attack, the security device (i.e., Snort) successfully detects this attack. And then NETSECVISOR enforces flow rules to drop all packets from Client 1. We capture screen images to show this operation. As shown in Figure 11, 192.168.1.12 is the attacker (Client 1), 192.168.1.11 is the target (Client 2), and 192.168.1.14 is the NIDS (Snort).

As shown in this simple case study, our prototype is user-friendly and tenants can easily create their security rules. Compared with Amazon EC2 Security Group [2], we have very similar user APIs and commands. However, tenants have more options of device types, traffic types and response actions when using NETSECVISOR. One can also define some default security policies for tenants if needed.

VII. LIMITATION AND DISCUSSION

There are several limitations in our current work. First, there could be some cases that NETSECVISOR cannot generate



(a) Generation Time Measurement (b) Network Cost Measurement

Fig. 8: 6-router topology

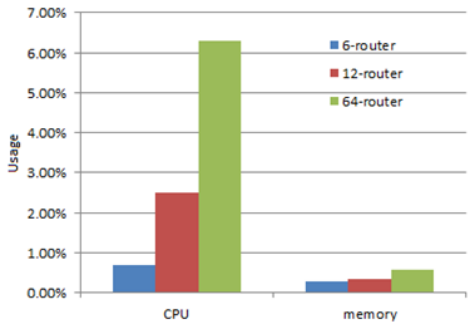


Fig. 9: CPU and Memory Overhead

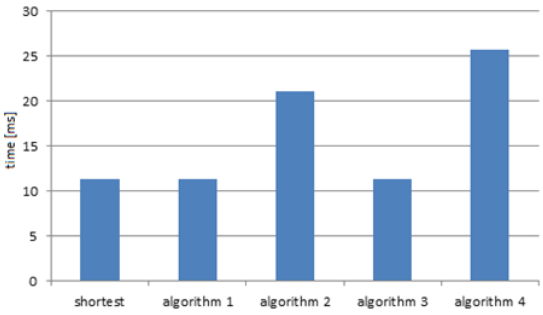


Fig. 10: Response Time

```
04/01-20:29:43.476919  [**] [1:276:5] DOS Real Audio Server [**] [Classification
: Attempted Denial of Service] [Priority: 2] {TCP} 192.168.1.12:38965 -> 192.168
.1.11:7070
04/01-20:29:43.485018  [**] [1:277:5] DOS Real Server template.html [**] [Classi
fication: Attempted Denial of Service] [Priority: 2] {TCP} 192.168.1.12:56374 ->
 192.168.1.11:7070
04/01-20:29:43.493007  [**] [1:278:5] DOS Real Server template.html [**] [Classi
fication: Attempted Denial of Service] [Priority: 2] {TCP} 192.168.1.12:50983 ->
 192.168.1.11:8080
04/01-20:29:43.498660  [**] [1:281:5] DOS Ascend Route [**] [Classification: Att
empted Denial of Service] [Priority: 2] {UDP} 192.168.1.12:19787 -> 192.168.1.11
:9
04/01-20:29:43.518891  [**] [1:1605:6] DOS iParty DOS attempt [**] [Classificati
on: Misc Attack] [Priority: 2] {TCP} 192.168.1.12:55476 -> 192.168.1.11:6004
04/01-20:29:43.526409  [**] [1:1545:8] DOS Cisco attempt [**] [Classification: W
eb Application Attack] [Priority: 1] {TCP} 192.168.1.12:9250 -> 192.168.1.11:80
```

(a) Snort Alert Message

```
[Admin] ------------------------------
[Admin] receive new packet from 192.168.1.12
Running algorithm 1
[Admin] ------------------------------
[Admin] receive new packet from 192.168.1.14
Receive message from passive mode device
Alert from 192.168.1.14:
Dangerous traffic from 192.168.1.12
Drop packets from 192.168.1.12
Alert from 192.168.1.14:
Dangerous traffic from 192.168.1.12
Drop packets from 192.168.1.12
```

(b) Response Strategy

Fig. 11: A Case Study

routing paths. For example if a network administrator specifies two security devices that cannot communicate with each other, our algorithms will fail in generating paths. However, NETSECVISOR can still show a warning message for this physically impossible routing path. Second, NETSECVISOR may suffer from tenants' mistakes or even malicious tenants if they register misconfigured or malicious policies. NETSECVISOR increases the trusted computing base (TCB) of the network controller (POX in our current implementation). We note that it is possible network controllers such as POX contain vulnerabilities that can be exploited by attackers to change the policies/flow rules. This is a separate research direction that worth further investigation from the research community (one of our future work). This issue is clearly out of the scope of NETSECVISOR and future work is needed in this area. Third, currently we only test our system in a small research environment with no more than 64 switches. In our future work we plan to evaluate in a larger-scale environment (e.g. GENI, enterprise clouds).

In this paper, we mainly focus on the situation when the security monitoring needs only a small number of security devices. When dealing with a set of various devices, we need to chain the services, which may involve more issues such as the order of services in chaining, the potential routing conflict/confusion for different devices. We plan to solve these interesting research problems in the future.

VIII. CONCLUSION

This paper introduces a concept of network security virtualization (NSV) that can virtualize security resources/functions and provide security response functions from network devices when necessary. We implement a new prototype system, NETSECVISOR, to demonstrate the utility of NSV. We have evaluated this prototype system in both virtual networks and

commodity switches. With promising results, we believe that NSV is a right step towards building more secure network environments efficiently and effectively.

REFERENCES

[1] Amazon. Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/.

[2] Amazon. Amazon ec2 security groups. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html.

[3] Jeffrey R. Ballard, Ian Rae, and Aditya Akella. Extensible and Scalable Network Monitoring Using OpenSAFE. In *Proceedings of USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking*, 2010.

[4] T. Benson and et al. Cloudnaas: a cloud networking platform for enterprise applications. In *In Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[5] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *LECTURE NOTES IN COMPUTER SCIENCE*, pages 18–38. Springer-Verlag, 2001.

[6] ETSI. Network function virtualization. http://portal.etsi.org/NFV/.

[7] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[8] Aaron Gember, Robert Grandl, Junaid Khalid, and Aditya Akella. Design and implementation of a framework for software-defined middlebox networking. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[9] Glen Gibb, Hongyi Zeng, and Nick McKeown. Outsourcing network functionality. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012.

[10] P. Gill and et al. Understanding network failures in data centers: measurement,analysis, and implications. In *In Proceedings of ACM SIGCOMM*, 2011.

[11] Google. Sdn research at google. http://research.google.com/pubs/Networking.html.

[12] Victor Heorhiadi, Vyas Sekar, and Michael K. Reiter. New Opportunities for Load Balancing in Network-Wide Intrusion Detection Systems . In *Proceedings of ACM CoNEXT*, 2012.

[13] IETF. Ospf. http://tools.ietf.org/html/rfc2328.

[14] Juniper. Juniper networks: Software defined networking. http://www.juniper.net/us/en/dm/sdn/.

[15] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *The Symposium on Operating Systems Design and Implementation (NSDI)*, 2010.

[16] David G. Luenberger and Yinyu Ye. Transportation and Network Flow Problems. In *Linear and Nonlinear Programming*, November 2010.

[17] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. In *Proceedings of ACM SIGCOMM Computer Communication Review*, April 2008.

[18] Mininet. Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet/.

[19] Michael J. Freedman Minlan Yu, Jennifer Rexford and Jia Wang. Scalable flow-based networking with DIFANE. In *In Proceedings of ACM SIGCOMM*, August 2010.

[20] Matthew L. Meola Michael J. Freedman Jennifer Rexford Nate Foster, Rob Harrison and David Walker. Frenetic: A High-Level Langauge for OpenFlow Networks. In *Proceedings of ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2010.

[21] Ankur Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *ACM WREN*, 2009.

[22] NetworkComputing. Verizon, partners push openflow, sdn. http://www.networkcomputing.com/next-gen-network-tech-center/232900451.

[23] ON.Lab. Open network operating system. http://onlab.us/tools.html.

[24] OpenFlow. Innovate Your Network. http://www.openflow.org.

[25] Pantou. Pantou: Openflow 1.0 for openwrt. http://www.openflow.org/wp/openwrt/.

[26] Phillip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN'12)*, August 2012.

[27] POX. Pox controller. https://openflow.stanford.edu/display/ONL/POX+Wiki.

[28] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of ACM SIGCOMM*, 2013.

[29] S. Raza, Guanyao Huang, Chen-Nee Chuah, S. Seetharaman, and J.P. Singh. Measurouting: A framework for routing assisted traffic monitoring. In *INFOCOM, 2010 Proceedings IEEE*, 2010.

[30] Vyas Sekar, Ravishankar Krishnaswamy, Anupam Gupta, and Michael K. Reiter. Network-Wide Deployment of Intrusion Detection and Prevention Systems. In *Proceedings of ACM CoNEXT*, 2010.

[31] J. Sherry and et al. Making middleboxes someone elses problem: Network processing as a cloud service. In *In Proceedings of ACM SIGCOMM*, 2012.

[32] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012.

[33] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A robust, secure, and high-performance network operating system. In *Proceedings of the 21th ACM Conference on Computer and Communications Security (CCS)*, 2014.

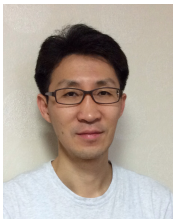
[34] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[35] Seungwon Shin and Guofei Gu. CloudWatcher: Network Security Monitoring Using OpenFlow in Dynamic Cloud Networks (or: How to Provide Security Monitoring as a Service in Clouds?). In *In Proc. of NPSec12, co-located with IEEE ICNP12*, 2012.

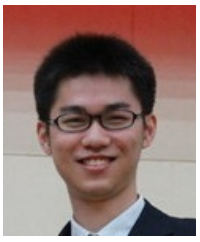
[36] Seungwon Shin, Phil Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. Fresco: Modular composable security services for software-defined networks. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[37] Andreas Voellmy and Paul Hudak. Nettle: Functional Reactive Programming of OpenFlow Networks. In *Yale University Technical Report*, 2010.

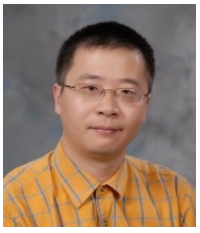
[38] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.



Seungwon Shin is an Assistant Professor in School of Computing at KAIST. He received his Ph.D degree in Computer Engineering from the Electrical and Computer Engineering Department at Texas A&M University (advisor, Professor Guofei Gu and Professor A. L. Narasimha Reddy). His research interests include Software Defined Networking(SDN), cloud network and botnet detection.



Haopei Wang is a Ph.D student in the Department of Computer Science and Engineering at Texas A&M University. His current research mainly focuses on Software Defined Networking (SDN) and network security. He is working in the SUCCESS (Secure Communication and Computer Systems) Lab, and his advisor is Dr. Guofei Gu.



Guofei Gu is an associate professor in the Department of Computer Science & Engineering at Texas A&M University. Before joining Texas A&M, he received his Ph.D. degree in Computer Science from the College of Computing, Georgia Tech, in 2008. He is currently directing the SUCCESS (Secure Communication and Computer Systems) Lab at TAMU.